Adapting Active Objects to Multicore Architectures

Ludovic Henrio, Fabrice Huet, Zsolt István, Gheorghen Sebestyén

Adapting Active Objects to Multicore Architectures

Ludovic Henrio Fabrice Huet
INRIA – CNRS – I3S – Univ Nice Sophia Antipolis, France
Email: {ludovic.henrio,fabrice.huet}@inria.fr

Zsolt István Gheorghe Sebestyén
Technical University of Cluj-Napoca, Romania
Email: zsolt.istvan@gmx.net, gheorghe.sebestyen@cs.utcluj.ro

Abstract

There are several programming paradigms that help programmers write efficient and verifiable code for distributed environments. These solutions, however, often lack proper support for local parallelism. In this article we try to improve existing solutions for providing a distributed, highly parallel framework that is easy to program. We propose an extension to the active object programming model which optimizes the local performance of applications by harnessing the full computing power of multi-core CPUs. The need for explicit locking mechanisms is reduced by the addition of meta-information to the methods in the source code. This paper describes this language-independent meta-information, and the way we intend to use it for parallelizing execution inside an active object.

Keywords

parallel programming paradigm and API, concurrent and distributed computing, active objects

1. Introduction

Even though there are several frameworks that are widely used in relation with distributed applications (OpenMP, RPC, Java RMI), these deal only with communication transparency, and do not hide all details of the distributed nature of the application. As a result, programmers have to deal with several low level aspects such as remote object registry setup and communication channel management. This gave rise to a new generation of frameworks that make all aspects of object distribution transparent to the user. Such high-level programming models with strong properties turned out to be crucial for programming safe large-scale applications.

Some of these frameworks are based on the actor paradigm ([1], [2], [3]). Actors are entities that are defined by their behavior and that communicate strictly by message passing, therefore are decoupled completely from each other [3], [4]. This model has been quite successful in several domains of distributed programming [5]. Active objects are inspired by actors, but they are closer to object oriented paradigms [6], [7], [8]. They are parts of a design pattern that decouples method execution from method invocation [9]. The main goal of active objects is to achieve global concurrency with the help of asynchronous communication and internal scheduling mechanisms for request handling.

Active objects, just as actors, are mono-threaded entities and therefore suffer from the same limitations regarding local parallelism. Taking into consideration the widespread popularity of multi-core processors and the trend of increasing the number of cores, any framework that does not fully utilize the multithreading capacities of multi-core architectures will seem deprecated. On the other hand, it is rather hard to deal both with the application logic and with the clearly orthogonal task of concurrent synchronization at the same time. Even if concurrent code is supposed to improve the performance of an application, if it is unwisely written it can introduce race-conditions, which make development and testing difficult [10], [11].

Our approach consists in extending the active object paradigm to support multi-threading of active objects while decoupling the logic flow from synchronization. We provide the user with a solution in which parallelism is transparent, just as the distributed nature of the application. The principles of multi-active objects can also be applied to all the actor-like frameworks, which could then benefit from a better support for local parallelism. One crucial requirement we impose ourselves is to keep the programming language easy to use for programmers who are not necessarily expert in concurrent programming. We thus decided to allow the programmer to annotate methods corresponding to entry points of the active object with information regarding parallelism. We use this information to run several method executions in parallel. Annotations

should be simple enough so that the programmer gives only high-level information, that we can use to "schedule" the parallel execution of several threads. This paper presents:

- A set of annotations to provide multi-threading for active objects, while keeping backward compatibility with standard active objects,
- A way to use those annotations to synchronize multi-threading inside active objects,
- An evaluation of our solution, intended at showing that it is adequate, i.e.: it is easy to use and has good performances.

This paper is organized as follows. In Section 2 we present the Active Objects and the ProActive framework, its strengths, and its shortcomings. Section 3 describes our parallel and distributed solution. In Section 4 we present an application illustrating our approach. Section 5 is a short presentation of related works. Section 6 concludes the paper.

2. Background and Objectives

All languages based on principles inspired from actors suffer from the same limitation concerning local parallelism. Indeed, actors are very efficient and very easy to program when it comes to distribution: actors abstract away the notion of distribution, remote communications occur in the form of messages, which are enqueued and treated by the receiver. This way different computing entities are strongly decoupled and synchronization only concerns the reception of messages. As a result, the programmer does not have to deal with data race-conditions. Unfortunately, when it comes to local parallelism, this programming model is far from being efficient because it entails a lot of data copy between actors while direct memory access would be faster. Some of the actor-like paradigms can be tweaked such that local activities are multi-threaded, but in that case, the programmer loses the benefits of a well-designed programming model, and has to face complex synchronizations and data race-conditions. Having a programming model that would mix the easiness of programming provided by actor-like languages with efficiency of local parallelism would be a great contribution to the domain of distributed and parallel computing.

ASP [12] is an object calculus inspired from an actor-like paradigm. It relies on asynchronous, future based communication between remote processes. Using a future [6], [13] as a placeholder for the result the caller can carry on executing after performing a call, as long as it does not need the actual result value. In ASP's Active Object model each active object maps

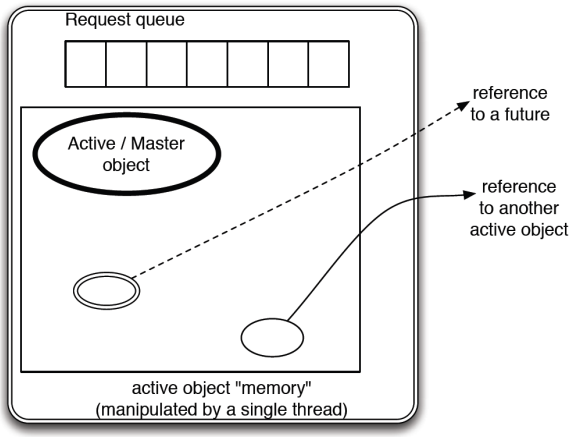


Figure 1. ASP Active object – simplified model

to a single thread of execution and has an internal queue for handling requests (Figure 1). When a request arrives, it is put into the object's queue and at the same time a future is created on the caller side. When a request has been served the active object replaces the future at the caller's side with the result of the call. In ASP no data can be shared between active objects directly, and as a consequence, parameters of remote method calls are deep-copied before transmission. At a given moment there could be several requests arriving simultaneously to active objects, however, concurrency is limited to the request queue of each object. Therefore the programmer is relieved from writing code that deals with synchronization, i.e. there is no need for definition of mutual exclusion areas, or usage of locks and other synchronization mechanisms. This strict programming model of ASP, which is based on sequential activities not sharing memory, helps in verifying the correct behavior of programs using formal methods [14].

ProActive is a distributed programming framework that implements ASP calculus in the Java programming language. It is a clean and straightforward implementation of the above presented model with a nice support for large-scale distribution (e.g. on Grids and Clouds), this is the reason why we chose it as the basis for our implementation. In addition, it provides object migration and has a built-in fault tolerance mechanism.

Unfortunately some applications are very difficult to write in ProActive's underlying model. For example, having re-entrant code in a pure sequential active object model is impossible. The problem with re-entrant calls is that once an active object is waiting for itself (indirectly) to produce a result, a deadlock occurs. To overcome these limitations, developers usually have to bypass the rules imposed by the model, making several

```
@DefineGroups({
  @Group(name="GroupF", selfCompatible=true),
  @Group(name="GroupB", selfCompatible=false)
})
@DefineRules({
  @Compatible( { "GroupF", "GroupB" } )
})
```

(a) Groups and rules

```
@MemberOf("GroupF")
public int foo_1() {...}

@MemberOf("GroupF")
public int foo_2() {...}

@MemberOf("GroupB")
public int bar() {...}
```

(b) Membership

Figure 2. Proposed annotations

features of the framework, like migration, unusable. Even more importantly such programs cannot rely anymore on verification tools that depend on the strict adherence to the programming model. Our solution seeks to obliterate the need for the circumvention of "ASP/ProActive rules". More precisely, we suggest to evolve the programming paradigm such that more programs can be expressed and higher efficiency can be achieved. The new programming rules are to be restrictive enough to enable the verification of program behavior by formal methods, and the design of new mechanisms for dealing with distributed objects (migration, fault-tolerance, etc.).

3. Proposal

The simplest solution for achieving multi-threaded active objects would be to immediately serve each request inside a new thread. Unfortunately, this solution is not adequate. Firstly, we would end up with unwanted concurrency, which introduces harmful data races and is difficult to program. Secondly, this breaks some of the basic properties of ASP, like the consistency between request arrival and serving order.

Another relatively simple solution, inspired by how web servers work, would be to use one thread per caller. This seems a good idea at first, because it combines parallel execution with apparently unchanged active object behavior when viewed by the "clients". Unfortunately, it is unsuitable for practical use in the general case, because this approach is adapted to stateless web servers, and cannot be used to implement stateful active objects. Since these active objects interact by altering the state of each other, either the data has to be duplicated, or one has to face unforeseen data race-conditions.

In order to create a truly useful multi-active object model, we decided to reason in terms of request compatibility. Compatibility of two requests means that running them in parallel either a) does not result in any data concurrency, or b) is expected by the programmer (i.e. he/she manages his/herself the data concurrency). In the second case it is supposed that the data in question will be protected in the code (with locks, mutual exclusion blocks, etc.). Since static analysis is out of the scope of this paper, we trust the programmer to define the compatibility rules among methods correctly. These rules can be thought of as contracts between the programmer and the runtime environment, in which the programmer allows the framework to run several methods in parallel. Whenever the programmer specifies that parallelism is harmful, he/she can rely on the runtime environment for assuring the safety of execution. In essence, the runtime will provide as much parallelism as possible, unless the programmer has stated otherwise.

3.1. Annotations

Method compatibility could be expressed in many different ways. We chose an approach where the programmer explicitly states the compatibility between methods, and the mutual exclusion is then deduced. Although being somehow similar to the use of the *synchronized* keyword in Java, it has more semantic flexibility. While the *synchronized* keyword is used to restrict parallelism between several methods, our approach allows for both a restrictive and permissive reasoning.

Obviously, pairwise compatibility relations for a high number of methods could easily become too complex to declare and to maintain. Therefore, we introduce the notion of groups to express compatibility relations on sets of methods rather than on individual methods. A group gathers methods that perform a similar task, thus manipulate the same data. Unless specified otherwise, methods inside a group are mutually incompatible. These groups not only have the goal of reducing the amount of added meta-data, but also help in the logical structuring of an application, since methods working on the same data set can be most probably collected into the same group. To specify which groups can run concurrently, a set of compatibility rules is given by the programmer.

To create the groups and to specify rules, the programmer will use annotations inside the source code placed at the beginning of classes. A group is defined by its name, that acts as an identifier used

```
method runActivity() {
  while (true) {
    serve(requestQueue.removeFirst());
  }
}
```

(a) Current FIFO

```
method runActivity() {
 while (true) {
   if (compatible(requestQueue.peekFirst(),
        activeRequests)) {
    parallelServe(requestQueue.removeFirst());
} } }
```

(b) Multi-active

Figure 3. Pseudo-code of service loops

in compatibility rules. If the methods contained in the group can be executed in parallel, its optional *selfCompatible* property can be set to true. Rules define sets of groups that are all compatible with each other. In the example shown in Figure 2a we create two groups and define their relationship as follows:

- Methods which are members of *FooGroup* can run concurrently because they are self compatible.
- Methods in *BarGroup* are mutually exclusive.
- Any method from *FooGroup* can run concurrently with a method from *BarGroup*.

The membership to a group is specified using an annotation written directly before a method (Figure 2b).

An advantage of using these annotations is that the amount of work needed to define rules does not depend on the size of a class, but only on the number of groups. If the complexity of performing this operation would grow exponentially with the number of exposed methods, this approach would not be practical in real-world applications.

Safely executing a multi-threaded active object then consist in serving a request in parallel with the others *if no incompatible request is currently being served*, to prevent race-conditions.

A potential drawback of the annotation-based approach is that, since meta-data is contained inside the Java classes, once they are compiled it can not be changed, and it is not possible to modify the compatibility of methods at runtime. This limitation can be addressed using scheduling policies presented below.

3.2. Multi-active scheduling

In the previous section we introduced the idea of compatibility annotations for active objects. We will proceed by presenting how this information can be used in the context of the ProActive framework to enable multi-threaded local execution.

In the ProActive framework each active object has to implement a method called `runActivity` that constitutes its life-thread. Most of the time this method is provided by the ProActive framework, and does not have to be implemented by the programmer. Since active objects can execute only a single request at a time, the default policy is to handle requests in a first-in-first-out manner.

Figure 3a shows the `runActivity` method provided by ProActive by default. The `serve` method takes a request as parameter and executes it in the context of the caller's thread. As a result, the service loop will return only when the request is served. This way (assuming that `removeFirst()` blocks if the queue is empty), there is no need for more logic inside the loop. To be able to serve several requests in parallel, a mechanism is needed to move their execution to secondary threads, started from the main loop. We introduce `parallelServe`, a non blocking method which will execute a request using a different thread. Figure 3b shows a new version of the `runActivity` method which executes requests concurrently if they are compatible. A list of currently executing requests is maintained (`activeRequests`) and the first request in the request queue is checked for compatibility, and served, if possible.

In the actual implementation this loop is hidden from the user inside the scheduling[1] class. The scheduler is a policy based one and it provides two predefined policies (or strategies): multi-active and FIFO. The first aims at starting as many requests in parallel as possible, while maintaining the relative order of their arrival at serving time. The second strategy exists for compatibility purposes, and can be used to reproduce the classic single-active behavior. Besides these two predefined policies we also expose a complete API which the programmers can use to write customized policies, e.g. one which limits the maximum number of parallel threads inside an active object.

The Scheduling API can be split into two parts. One that deals with method compatibility and one that exposes the internal state of the scheduler. The first contains methods for verifying the compatibility of two (or more) requests or method names. The second gives access to the request queue and the set of already executing requests inside the active object. To simplify

1. note that the term "scheduling/scheduler" is used in this paper to mean: "definition of a multi-threading service policy" and is not directly related to the vast research area related to the design of efficient schedulers for distributed computing.

```
Given: G = (V ,E).
procedure FB(V)
    pick pivot v ∈ V ;
    F := Forward(v);
    B := Backward(v);
    report F ∩ B;
    in parallel do
        FB(F \ B);
        FB(B \ F);
        FB(V \ (B ∪ F));
    end parallel
end FB
```

Figure 4. Reference SCC search algorithm

```
@DefineGroups({
  @Group(name="Forward", selfCompatible=true),
  @Group(name="Backward", selfCompatible=true) })
@DefineRules({
  @Compatible({ "Backward", "Forward" }) })
public class GraphWorker implements RunActive{
    @MemberOf("Forward")
    public Set<Integer> markForward( ... ) {...}
    @MemberOf("Backward")
    public Set<Integer> markBackward( ... ) {...}
    ...}
```

Figure 5. Annotated code from the worker

the policy-writing, a policy is defined as a function, which takes as input the compatibility information and the scheduler state, and produces as an answer the list of requests that can be started right away. The internal state of the scheduler is guaranteed not to change while executing a policy, so the code of the policy does not have to deal with synchronization.

3.3. Ensuring backward compatibility

The design of the extension for the framework was done with backward compatibility in mind. This is why, in case there is no compatibility information added to a method, it will be considered incompatible with all the other methods. This maps exactly to the behavior of the current version of the ProActive framework. It also protects from accidental, unwanted parallelism, making the internal parallelization of an already existing application much easier – it can be done incrementally, by adding meta-information to more and more methods.

4. Experiment

4.1. Parallel and Distributed SCC search algorithm

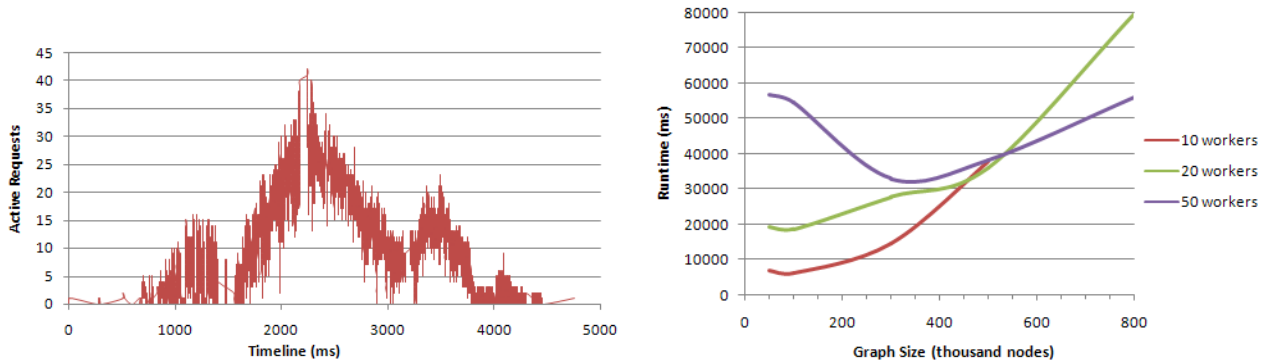
To verify that our proposed model is practical for realistic algorithms, we chose to implement a strongly connected component[2] (SCC) search algorithm, described in [15]. This algorithm exhibits a high degree of parallelism and relies on recursive calls. It is thus difficult to implement it without heavy modifications in a single-threaded framework. The main purpose of this experiment was to show that, with our proposed extension, implementing such an algorithm can be done without changing its underlying logic.

2. A sub-graph of a directed graph is called strongly connected if there is a path from each of its vertices to every other vertex. The strongly connected components of a directed graph are its maximal strongly connected sub-graphs.

The implemented algorithm searches for strongly connected components in graphs using a divide-and-conquer approach (Figure 4). To find such sub-graphs, the algorithm will start from arbitrary vertices (pivots) and calculate the biggest strongly connected component containing the pivot vertex. After this, the search will continue on the remaining parts of the graph in parallel. The *marking* step is very important in this algorithm and consists in finding all the vertices reachable from a given vertex in the graph, if it is a forward marking, or all vertices that can reach the given one, if it is a backward marking. The first source of local parallelism we exploited is the fact that forward and backward marking can be performed in parallel without conflicting with each other. In addition, marking steps started at different branches of the algorithm can execute in parallel, yielding even higher levels of local parallelism.

Another source of parallelism is that the graph is distributed among several workers. Each worker "owns" only a part of the original graph and has references to its neighbors. In case a SCC is not completely contained in the owned sub-graph of a worker, the marking procedure will propagate to neighboring workers. This way, a worker can be involved at the same time in several marking operations, started at different nodes, and at different branches of the main algorithm. In our implementation, the workers are multi-active objects. A central coordinator uses their exposed methods for graph loading and starting the marking operation.

Taking the previous observations into account, we have annotated the Java code of the workers as shown in Figure 5. Two groups, `Forward` and `Backward`, are created and the methods performing the marking (`markForward` and `markBackward`) are added to the corresponding one. This allows a worker to execute concurrently these methods. The groups are `selfCompatible` because it is possible to execute multiple markings of different origin.



(a) Evolution of the number of parallel serves in a multi-active object

(b) Runtime vs. number of workers in a distributed setting

Figure 6. Experimental results

4.2. Performance

The experiments were carried out on the Grid5000[3] platform, on nodes equipped with two *Intel Xeon E5520* or *Intel Xeon E5420* quad-core processors, running at 2.27GHz, respectively 2.5GHz. The nodes were connected via Gigabit Ethernet connections.

First, we have measured the number of requests simultaneously executing on a worker (Figure 6a) while processing a graph of size 10000 with 25 strongly connected components. As expected, at several times during execution, local parallelism was quite high (41 simultaneously served requests).

During another set of experiments, to show that the overall performance of the application is satisfactory, we have run our application for different sizes of graphs having 10, approximately equally sized SCCs, for three different number of workers (10, 20 and 50). The results depicted in Figure 6b show that a higher number of workers yields a better overall performance on large enough graphs. As usual, the ratio of computation/communication explains the shape of the graph. For small graphs, the local processing time is much lower than the communication time, leading to low performance when the number of workers is too high. This is expected because the algorithm was not optimized, as our purpose was to limit the modification of the sequential code. Above 500 thousand nodes, 10 workers are not sufficient and the speedup obtained by adding workers is relatively good.

This experiment has shown that it is possible to harness both distributed and local parallelism with multi-active objects in large-size experiments, without modifying the core of the underlying algorithm. We consider beneficial the fact that algorithms, which until now would have been very complicated to program using a single-threaded active object based framework, such as ProActive, can be implemented quite efficiently using multi-active objects.

3. Acknowledgment: Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

5. Related Work

We compare our work with the most relevant approaches, which harness concurrency or distribution by providing high-level constructs. We particularly focus on languages that rely on active objects, or code parallelization based on annotations in the source code.

X10 [16] is a programming language under development that adopts a fairly new model, called partitioned global address space (PGAS). In this model, computations are performed in multiple places (possibly on various computational units) simultaneously. Data in one place is accessible remotely, and is not movable once created. Computations inside places are locally synchronous, but inter-place activities are asynchronous. This decouples places and ensures global parallelism. While this model seems fundamentally different from ours, the possibilities provided by the two are comparable. Places can host multiple activities, resulting in a similar service to what multi-active objects offer. One thing that the language does not ensure, is encapsulation, which even though might be costly in terms of performance, is known to ease the development of applications [17]. We try to allow for both worlds at the same time: offering powerful local

concurrency, while ensuring strong encapsulation of each multi-active object.

AmbientTalk [18] implements the ambient-oriented programming paradigm, which brings a novel approach to mobile network based applications. At its core however (similarly to [4], [6], [7]) it relies on active objects. These are called AmbientTalk actors, and are single-threaded. As a consequence, they have all the drawbacks (except the re-entrance problem) of single-active objects we presented throughout the paper. Considering the similarity between AmbientTalk actors and ProActive active objects, adapting our approach to the AmbientTalk framework seems particularly easy.

The Creol [7] language also advocates the active object paradigm as the model of distributed computation and it features futures with some form of multi-active objects. Distribution principles in Creol are quite similar to ASP and ProActive. However, Creol [7] handles multi-threading much differently from us. First, in Creol, only one thread is active at any time, which prevents race-conditions but does not allow local parallelism inside an active object. In Creol, similarly to our solution, the granularity of the thread creation is the request service (the handling of a remote method invocation), but one thread is created for each request that is to be served. Moreover, the programmer has to handle threads explicitly, placing waiting points in his code to let other threads run. Even though the two programming models have several similarities, and ideas developed in one could be used in the other, our approach features much greater transparency. In our opinion, this simplifies greatly the programming.

JAC [19] is an extension of Java that introduces a higher level of concurrency and separates thread synchronization from application logic in a declarative fashion. JAC relies on a custom precompiler and declarative annotations, in form of Javadoc comments placed before method headers. These annotations show several similarities with our approach, and it could be possible to express our multi-active objects in JAC. However, even for "simulating" the simplest active objects, the added annotations will become very complex. In our opinion multi-active objects offer a simpler annotation system and a higher synchronization logic encapsulation for the programmer. On the other hand, JAC's inheritance model is well defined, and useful for our annotations also.

JPPAL [20] is an annotation language for parallel programming. It is inspired by the semantics of OpenMP and implemented as a library of aspects in AspectJ which ensure instruction-level parallelism. [20] argue that performance is close to the one obtained with Java threads while their approach requires much less programming effort. Although, the original code has to be refactored (method headers and return types), and in our opinion this is an undesirable property, as in some ways it removes transparency.

JConqurr [21] aims at similar goals as JPPAL, but it is a programming IDE that generates parallel source code, based on directives placed in the code. The main issue with this approach is that debugging is particularly hard, as the sequential and parallel version of code can be quite different.

Pluggable parallelization [22] is a solution for Java that takes after an other well-known library for parallelism: MPI. Code is plugged into the original source with the help of *C++*-like templates. Even though this solution is most suitable for SPMD applications, it is possible to create non-SPMD ones with the help of remote objects and asynchronous method calls. This concept is promising in terms of performance and separation of concerns, but in our opinion, its template system is rather complex and unusual in Java.

While the previous ([20], [21], [22]) deal with instruction level parallelism, PAL [23] targets method level parallelization, in Java. It relies on code annotations to ensure load-time local and distributed parallelization. To prevent data-races, it does not allow for parallel methods to access class fields. Unfortunately, this restriction can be costly in terms of performance due to data-copying overhead, and in terms of code clearness, due to forced modifications in the application logic.

This section showed several works which improve the classical active object paradigm, or provide meta-information-based parallelism. Unfortunately, none of them quite succeeded in providing a model that is easy to program and has a high-level of abstraction.

6. Conclusion

In this paper we have presented a way to extend active objects to offer support for local parallelism and reentrant calls. Based on the use of annotations to indicate the possibility of methods to run in parallel, our approach reduces the need for explicit locking. It brings a high degree of parallelism, is extensible, and compatible with legacy active objects. On the programmer side, the complexity is minimal: only some simple compatibility annotations have to be added inside the code. We have also proposed an API which can be used to implement custom scheduling policies to improve the performance of an application.

A graph search algorithm has been implemented for testing, which proves the ease of use of our proposal. Starting from the standard algorithm, only a few modifications were required to obtain a distributed version

featuring high levels of local parallelism. This version was then easily deployed to a cluster of 50 nodes.

We are now interested in a policy which allows a request to be served not only if it is the first in the queue but also if the requests placed before it in the queue are compatible with it. This policy will ensure maximum possible parallelism inside the active object while, at the same time, maintaining causality relationship among the results that would exist in the sequential active objects. In the future we want to prove the correctness of this policy and experiment with it.

References

[1] G. Agha, "An overview of actor languages," in *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*. ACM, 1986, pp. 58–67.

[2] G. Agha and C. Hewitt, "Actors: A conceptual foundation for concurrent object-oriented programming," in *Research directions in object-oriented programming*. MIT Press, 1987, pp. 49–74.

[3] J. Armstrong, "Erlang-A survey of the language and its industrial applications," in *In Proceedings of the symposium on industrial applications of Prolog (INAP96). 16–18*. Citeseer, 1996.

[4] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[5] U. Wiger and E. Ab, "Four-fold Increase in Productivity and Quality-Industrial-Strength Functional Programming in Telecom-Class Products," 2001.

[6] K. Taura, S. Matsuoka, and A. Yonezawa, "ABCL/f: A future-based polymorphic typed concurrent object-oriented language-its design and implementation," in *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*. Citeseer, 1994.

[7] E. B. Johnsen, O. Owe, and I. C. Yu, "Creol: A type-safe object-oriented model for distributed concurrent systems," *Theoretical Computer Science*, vol. 365, no. 1–2, pp. 23–66, Nov. 2006.

[8] D. Caromel, W. Klauser, and J. Vayssiere, "Towards seamless computing and metacomputing in Java," *Concurrency: practice and experience*, vol. 10, no. 11-13, pp. 1043–1061, 1998.

[9] R. Lavender and D. Schmidt, *Active object: an object behavioral pattern for concurrent programming*, 1996.

[10] P. Hansen, "Java's insecure parallelism," *ACM Sigplan Notices*, vol. 34, no. 4, pp. 38–45, 1999.

[11] E. Ábrahám-Mumm, F. de Boer, W. de Roever, and M. Steffen, "Verification for Java's reentrant multithreading concept," in *Foundations of Software Science and Computation Structures*. Springer, 2002, pp. 3–13.

[12] D. Caromel, L. Henrio, and B. Serpette, "Asynchronous and deterministic objects," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2004, pp. 123–134.

[13] B. Liskov and L. Shrira, "Promises: linguistic support for efficient asynchronous procedure calls in distributed systems," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. ACM, 1988, pp. 260–267.

[14] T. Barros, A. Cansado, E. Madelaine, and M. Rivera, "Model-checking Distributed Components: The Vercors Platform," *Electronic Notes in Theoretical Computer Science*, vol. 182, pp. 3–16, 2007.

[15] J. Barnat, J. Chaloupka, and J. Van De Pol, "Distributed algorithms for SCC decomposition," *Journal of Logic and Computation*, 2009.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005, pp. 519–538.

[17] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *Conference proceedings on Object-oriented programming systems, languages and applications*. ACM, 1986, p. 45.

[18] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented programming in ambienttalk," *ECOOP 2006–Object-Oriented Programming*, pp. 230–254, 2006.

[19] M. Haustein and K. Lohr, "JAC: declarative Java concurrency," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 5, pp. 519–546, 2006.

[20] J. S. E. Sousa, "Jppal: Java parallel programming annotation library," in *Proceedings of the 2010 AOSD workshop on Domain-specific aspect languages*, 3 2010.

[21] G. Ganegoda, D. Samaranayake, L. Bandara, and K. Wimalawarne, "JConqurr-A Multi-Core Programming Toolkit for Java," *International Journal of Computer and Information Engineering*, vol. 3, no. 4, 2009.

[22] R. Gonçalves and J. Sobral, "Pluggable parallelisation," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 2009, pp. 11–20.

[23] M. Danelutto, M. Pasin, M. Vanneschi, P. Dazzi, D. Laforenza, and L. Presti, "PAL: Exploiting Java Annotations for Parallelism," *Achievements in European Research on Grid Systems*, pp. 83–96, 2008.